Ceylan-Curry, a Haskell Cookbook

Contact: about (dash) curry (at) esperide (dot) com

Creation date: Tuesday, September 7, 2021

Lastly updated: Wednesday, September 7, 2022

Version: 0.0.1

Status: Work in progress

Dedication: Anyone wanting to discover the Haskell programming language.

Abstract: The purpose of this Curry cookbook is to help newcomers getting up to speed with functional programming as done based on the Haskell language.

The latest version of this documentation is to be found at the official Curry website (`http://curry.esperide.org`).

Table of Contents

Overview

The purpose of this Curry cookbook is to help newcomers getting up to speed with functional programming when relying on the Haskell language[1] for that.

More precisely, this cookbook is to summarise the various elements that we found useful to remember when wanting to program in Haskell. We hope that it may be useful whereas either one never really practiced that art or already forgot essential elements of it.

So the goal of Curry is to be quicker to read/browse than it would be to start the learning again from scratch, and never reaching the latter parts thereof (knowing that the learning curve of Haskell is unfortunately rather steep).

> **Note**
> As this cookbook is being written as we are in the process of learning Haskell, errors, misconceptions and epic blunders are bound to occur in this text; if you detect such issues, please contact us so that we can correct this document accordingly. Thanks in advance!

Cookbook Conventions

The ≡ character denotes here equivalence of expressions; ex: `2 * x` ≡ `x + x`.

For clarity, the formal characters like , and → are replaced by the one that you would type (namely \, >> and ->).

Concepts

Functional Programming (FP)

A programming style based on the application of functions (more information).

Arity

The (maximal) number of arguments expected by a function.

Expressions

- conditional ones: if/then/else (ex: `if n >= 0 then n else -n`)
- guarded: `|` can be read as "when (some condition is true)"; knowing that `otherwise = True` allows to define default clauses
- lambda expressions are just anonymous functions; ex: `\x -> 4 + 2*x`

[1] This cookbook is in some way a Haskell counterpart of what we did for Erlang, with the software stack whose first layer is Ceylan-Myriad.

Operators

Operator Precedence

Precedence allows to define the order of the operations.

If the precedence of `op1` is higher than the one of `op2`, then `x op1 y op2 z` shall be read as: `(x op1 y) op2 z`.

See the precedence table for all Haskell operators.

By using the `:info` GHCi command, the precedence levels of operators can be returned:

```
Prelude> :info +
type Num :: * -> Constraint
class Num a where
  (+) :: a -> a -> a
  ...
    -- Defined in ‘GHC.Num’
infixl 6 +
```

Operator Associativity

This determines how operators of the same precedence are grouped in the absence of parentheses.

Operators may be:

- **associative**, meaning the operations can be grouped arbitrarily
- **left-associative**, meaning the operations are grouped from the left: `2-3+4 = (2-3)+4`
- **right-associative**, meaning the operations are grouped from the right: `2^3^4 = 2^(3^4)`
- **non-associative**, meaning operations cannot be chained, often because the output type is incompatible with the input types

Operator Calls

Operators are just functions that can be called:

- either directly: `op1 x` or `op2 x y`; for example: `(-x)` or `(+) x 4`
- or, for the ones of arity 2, as infix operators: `x ‘op2‘ y` ≡ `op2 x y`

Some Operators of Interest

Calling functions: the " " (pseudo-)operator Being central in FP, **function application** is symbolised just by a space.

It behaves as the operator of the highest precedence; therefore `f a + b` ≡ `(f a) + b`

(rather than `f (a + b)`)[2].

[2] Another example: an expression that could be described informally as `f(a,b) + c.d` translates in Haskell as `f a b + c*d` or `((f a) b) + c*d"`.

This pseudo-operator is left-associative: `f g h` ≡ `((f g) h)`.

`f g x` corresponds to the application of a function `g` and of a variable `x` at `f` (i.e. `f(g,x)`); if wanting to express `f(g(x))`, rely on `f (g x)` or, even better, on `f.g x`.

Consing lists: the ":" operator This operator, named `cons` (for *construct*), allows to define lists by appending successively elements, starting from the empty list (`[]`; designated as `nil`).

The `:` operator is right-associative:

```
x:y:z:l  ≡  x:(y:(z:l))
```

Example:

```
[4,5,3] = 4:5:3:[]
```

Function Arrow: the "->" operator It allows to specify the datatypes involved in the type definition of a function.

This operator is right-associative:

```
A -> B -> C ->  ≡  A -> (B -> (C -> D))
```

Example:

```
-- mult :: Int -> (Int -> (Int -> Int))
mult :: Int -> Int -> Int -> Int

-- mult x y z = ((mult x) y) z
mult x y z = x*y*z
```

Composition: the "." operator The `.` operator applies to functions, and returns the composition of two functions as a single one: `(f . g) x = f (g x)`.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
```

Composition is associative: `f . (g . h) = (f . g) . h`.

Compositions allow to think in terms of functions and abstract out values.

An example to express the composition of a list of functions:

```
compose :: [a -> a] -> (a -> a)
compose = foldr (.) id
```

Non-Binding Function Application: the "$" operator The `$` operator is another way, besides the usual function application (denoted by a space) , of applying arguments to a function.

This is the operator of least precedence. It has been introduced in order to further avoid the use of parentheses:

```
a $ b op c  ≡  a (b op c)
```

When a `$` is encountered, the expression on its right is applied as the argument to the function on its left, as if a virtual parenthesis was opened there, and a closing one added at the end of the expression.

This operator is right-associative: `f $ g $ h` ≡ `(f $ (g $ h))`.

So this operator is the "opposite" of the base function application in terms of precedence and associativity - but otherwise is the same in terms of definition (including typing):

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Function

An association from a set of arguments to a set of corresponding results.

```
double x = 2*x

sum :: Num a => [a] -> a
sum []     = 0
sum (n:ns) = n + sum ns
```

A function may not be defined for all (well-typed) combinations of its arguments.

For example `head []` is to throw an exception.

A function is a value like the other datatypes (first-class citizen).

Thanks to the algebraic datatypes such as lists and tuples, a function may take and return an arbitrary number of values (thus including functions).

A function may be only **partially applied**: its last argument(s) may not specified in a given call. Then the value of the overall expression is a function of these remaining, unspecified arguments.

For example, if `f :: (Int,Float) -> Bool` then `f 3 :: Float -> Bool`.

By default functions are polymorphic, insofar as they will accept all types determined as compatible. For example `zip :: [a] -> [b] -> [(a,b)]` can handle any types for `a` and `b`.

Functions are defined from expressions and based on pattern matching, which operates on few possible patterns that are examined in turn, from top to bottom:

- function application, like in:

  ```
  (&&) :: Bool -> Bool -> Bool
  True && b = b
  False && _ = False
  ```

- tuple patterns, like in:

  ```
  fst :: (a,b) -> a
  fst (x,_) = x
  ```

- list patterns, like in:

  ```
  tail :: [a] -> [a]
  tail (_:xs) = xs
  ```

The ‘_’ character acts as a "wildcard" (it matches any value).

Note that a non-empty list is (a bit surprisingly) to be pattern-matched with `(x:xs)` and not `[x:xs]`. For example:

```
product :: Num a => [a] -> a
product [] = 1
product (n:ns) = n * product ns
```

The same name cannot be specified for two arguments to match when they are equal; a guard must be used for that. Finally pattern matching can operate on few possible patterns: function application, tuple and list ones, that's it.

Guards

Functions can be defined using guarded equations, to select which clause applies based on the first one from the top to evaluate to `True`. As Haskell can guarantee that these functions are pure, guards can be user-defined (rather than be taken in a limited selection of built-in guards).

For example:

```
abs | n >= 0    = n
    | otherwise = -n
```

Guards are also logical expressions to filter the values produced by earlier generators.

For example:

```
primes :: Int -> [Int]
primes n = [ x | x <- [2..n], prime x]
```

Currying

A function taking its arguments one by one (one at a time, from left to right), each time returning a function of a decremented arity, is said to be **curried**.

Such 1-arity functions are those directly modelled by the lambda calculus, an essential base of the functional languages.

As the function arrow operator (`->`) is right associative, the type of functions is preferably written as:

```
f :: TArg1 -> TArg2 -> ... -> TArgn -> TResult
```

This corresponds to the following actual type:

```
f :: TArg1 -> ( TArg2 -> (... -> (TArgn -> TResult))...)
```

Consequently, as their arguments are to be applied one by one from left to right, the “ “ operator (function application is represented by the space character) is right-associative:

```
f a1 a2 ... an  ≡  (((f a1) a2) ... an)
```

Functions can thus all be seen as curried ones, and can also be directly transposed as lambda ones; for example the following definitions define the same function:

```
add :: Int -> Int -> Int
add x y = x + y

add :: Int -> (Int -> Int)
add = \x -> (\y -> x + y)
```

In this latter form:

- the type specification and the definition of the function respect the same structure: `XXX -> (XXX -> XXX)`
- the function is designated (on the left of the `=` sign) without listing its arguments (we have `add =`, not `add x y =`)[3]
- if the purpose of the function is to return another one, its intent is clearer once expressed as a lamdba function

Side Effects

They correspond to all the consequences that the evaluation of a code (program, function) incurs besides its returned value.

Example:

- writing content on file, or on the screen
- reading from file or from the input devices (keyboard, mouse)
- changing the state of a value accessible from outside of the function
- sending a message to another process or through the network
- drawing a random value

These impure events are difficult to manage yet are generally necessary, as the purpose of a program is to trigger "interesting side-effects"; a strictly pure program would most probably have no actual use (except using time, processing resources and adding the possibility of failure).

Typing

Notion of Type

A type is a set of associated values.

`e :: T` means that expression `e` is of type `T`.

Haskell infers types at compilation (static typing), which prevents to discover many problems[4] at runtime.

Some expressions that could be successfully evaluated can nevertheless be rejected on type grounds (ex: `if True then 1 else False`), but in practice it is hardly a problem.

[3] This is certainly clearer yet, if no type specification is given, the arity of the function is only implicit then.

[4] Of course other problems may occur; for example `1 ‘div‘ 0` is well-typed yet its execution will fail.

Type Alias

Ex: `type Pos = (Int,Int)`

This does not introduce a new type; these are synonyms for Haskell.

Type declarations can have any number of parameters and be parametrised by other types, like in `type Assoc k v = [(k,v)]`.

Data Declarations

This introduces a new type; ex: `data Bool = False | True`.

`|` shall be read "or", and the values specified for the new type are called *constructors*. They start with an uppercase letter and a given constructor may not be used in more than one type.

Constructors may have arguments, in which case they are *constructor functions*:

```
data Shape = Circle Float | Rect Float Float
```

Here `Circle :: Float -> Shape`.

Such functions have no definition equations, and `Circle 1.0` is fully evaluated and cannot be further simplified.

The ordering on the constructors of a type is determined by their position in its declaration (ex: `False < True`; constructors with arguments are ordered lexicographically regarding these argments.

They may also be:

- parametrised: `data Maybe a = Nothing | Just a`.
- recursively defined: `data Nat = Zero | Succ Nat`

For example all sorts of trees can be defined with it:

- binary tree with content only in leaves:

  ```
  data Tree a = Leaf a | Node (Tree a) (Tree a)
  ```

- binary tree with content only in non-leaf nodes:

  ```
  data Tree a = Leaf | Node (Tree a) a (Tree a)
  ```

- binary tree with different types of content in leaf and non-leaf nodes:

  ```
  data Tree a b = Leaf a | Node (Tree a b) b (Tree a b)
  ```

- tree having any number of subtrees:

  ```
  data Tree a = Leaf a | Node a [Tree a]
  ```

Newtype Declarations

Introduces a new type provided that it has a single constructor with a single argument.

Ex: `newtype Nat = N Int`.

By comparison:

- `type Nat = Int` would not introduce a new type
- `data Nat = N Int` would also introduce a new type, but would incur runtime overhead

Why `data` is not automatically transformed by Haskell internally in `newtype` whenever applicable is unsure (for us).

Algebraic Datatypes

They are defined based on unions of values.

For example, in:

```
data MyFirstType = FirstValue | SecondValue | FirstConstructor T
```

`MyFirstType` is the *name* of the type. `FirstValue` and `SecondValue` are (constructor) *values*.

`FirstConstructor` is a constructor, and `T` is a *type name* (not a constructor).

Type and constructor names *can* be the same, as no ambiguity can occur.

Polymorphic types can be defined:

```
data MySecondType a = ThirdValue | SecondConstructor a Int
```

Basic Types

- monomorphic: `Bool`, `Char`, `String :: [Char]`, `Int`/`Integer`, `Float`/`Double`
- polymorphic: list, tuple, function

Char Example: `'a'`.

Tuple A tuple is a fixed-sized, possibly heterogenous container.

Example:

```
("I am a tuple", 2, True)
```

List

Description A list is an arbitrary-sized, homogeneous container.

Example:

```
l1 = [1.0, 7.0, 2.0]
l2 = []
l3 = [100..]
l4 = 1 : 2 : 3 : []
```

List Comprehension Example:

```
u s = [toUpper c | c <- s]
```

More information: [1].

Folds Like the `map`, the various folds encapsulate a classical recursion pattern.

`foldr` (`r` for right fold) evaluates from right to left, uses a right-associative operator and is directly recursive:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v []     = v
foldr f v (x:xs)  = f x (foldr f v xs)
```

`foldl` (`l` for left fold) evaluates from left to right, uses a left-associative operator and relies on an accumulator:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f v []     = v
foldl f v (x:xs)  = foldl f (f v x) xs
```

String A string is nothing but a list of (Unicode) characters: `String` ≡ `[Char]`.

Example: `"Hello!"` or `'a':'b':'c':[]`

A string cannot spread over multiple lines directly; the backslash (\) character is needed for that:

```
s = "This is a unique \
\line."
```

(any text between the two \ is ignored)

A newline is designated in a string by `\n`.

Function See function.

Type Class

A type class is a set of types that support a set of functions called `methods`.

For example the `Eq` class is to gather all types that can be compared for egality (or inegality), based on the following two methods:

```
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
```

A type may belong to one or more classes.

Class constraints can be specified when typing a function. For example:

```
(+) :: Num a => a -> a -> a
```

The type variable `a` must be here an *instance* of class `Num`.

A type respecting at least one of such constraints is said *overloaded*, as is the corresponding expression.

Well-known classes are: `Eq`, `Ord`, `Show`, `Read`, `Num`, `Integral`, `Fractional`.

Monad

Informal Descriptions

Various intuitive descriptions of a monad apply:

- a stateful datastructure representing some processing
- an abstraction, a generic concept allowing to structure programs generically and to unify in a functional way various problems (to structure them and favour separation of concerns)
- programmable ";" (semicolons whose effect can be defined)
- a structure allowing to express imperative traits in functional languages, to convey notions like exceptions or side-effects while preserving their purity
- a way of describing and composing impure expressions in a pure context: a monad is an expression still to be evaluated (potentially inducing then side-effects), yet able to be integrated in a pure context
- the code source of an imperative program that, once executed, returns a value of a specified type and that can be chained with other programs taking and returning such values; the language handles then the imperative programs themselves (ex: their source) rather than the values they return

More Formally

A monad M is a `(type, return function, bind function)` triplet with:

- a monadic constructor: `t -> M t`
- a function named `return` that, through the previous constructor, allows to obtain, from a value of type `a`, a value of monadic type `M a`: `return: t -> M t`
- a function named `bind` that allows to compose a monadic function[5] with others, represented by the `>>=` infix operator so that: `>>= :: M t -> ( t -> M u ) -> M u`

So `return` allows to convert a pure expression (not involving side-effects) of type `t` into an action (i.e. a transformation that may be executed) `M t` that is impure (with side effects) and deals with a value of type `t`. No opposite operation exists (that would transform a value of type `type M t` into one of type `t`, in the sense that side-effects cannot be rolled-back. A goal is thus to separate the pure parts of a program from its impure ones, which shall be minimised and isolated.

`Mt >>= f` (i.e. "bind Mt f") allows to apply the f function to the value of type t encapsulated in `Mt`; `>>=` drives the chaining of monadic functions (as such it handles functions, not values).

By composing `>>=` with `return`, any function `g :: t -> t` can be applied to a monad of type `M t` (here the types `u` and `t` in the definition of bind are the same).

[5] A monadic function is a function returning a monadic value.

These functions (`f` et `g`) are only aware of the value (`t`) encapsulated in the monad, not the monad (`M t`) itself.

So the developer composes a sequence of function calls (a pipeline) by chaining binds in an expression. Functions transform the values that they receive ; then the bind operator controls the returned monadic values (ex: it can enrich them outside of the view of these chained functions) and the next calls (ex: it can make them conditional).

So a monad of type `M t` is an algebraic datatype that derives from type `t`.

The elements of the monadic triplet shall respect following 3 axioms, `return` behaving like a neutral element from `>>=`:

- left composition by `return`, with: `(return x) >>= f` ≡ `f x`
- right composition by `return`, with: `m >>= return` ≡ `m`
- associativity of bind, with: `(m >>= f) >>= g` ≡ `m >>= \x . (f x >>= g)`

A sequence of actions (transformations that may be executed) may be combined into a single composite action thanks to the `do` notation.

Let's consider types `Vk` for k in `1..n` and `R`, and a function `f` of arity `n` and type `V1 -> V2 -> ... -> Vn -> R`.

```
do v1 <- a1
   v2 <- a2
   ...
   vn <- an
   return (f v1 v2 ... vn)
```

The `<-` operator in `v1 <- a1`, for `v1` a value of type `V1`, means that the action (transformation function) `a1` of type `M V1` (i.e., if `M` is `IO`, of type `WorldState -> (V1,WorldState)`) shall be *executed*, and that its result (of type `V1`) shall be stored in variable `v1`. Then the (pure) function `f` can be evaluated with these resulting `vk` arguments, and its returned value of type R is returned as an action of type `M R` (ex: `IO Int`).

As a consequence, the `do` notation returns a new action, i.e. a single composite transformation that may be executed.

Refer to this article for more information.

In a Nutshell

A monad `M` defines a type that represents a processing and is associated to two operators:

- `return`: to encapsulate a value of type `t` in this monad (resulting in a monadic value `M t`)
- bind, i.e. `>>=`: to compose monadic functions

Examples

The `IO` monad is probably the most well-known monad. `IO t` represents an imperative program taking no parameter and returning a value of type `t`.

For example:

- `IO ()` denotes, with the empty tuple `()`, a program returning no specific value (akin to `void` in some languages)
- the `getLine‘‘function is of type ‘‘IO String`, it returns the string entered on the keyboard by the user

A second example is the one of `Maybe`, as taken from this article
The Maybe-type (so here `M = Maybe`) is: `Maybe t ::  Just t | Nothing`
The Maybe-return is:

```
return :: t -> Maybe t
return undefined = Nothing
return 0 -> Just 0
```

The Maybe-bind is:

```
(>>=) :: Maybe t -> ( t -> Maybe u ) -> Maybe u
Nothing   ‘‘‘>>=‘‘‘ f = Nothing
(Maybe a) ‘‘‘>>=‘‘‘ f = Just( f a )
```

The Maybe monad allows to handle errors: as soon as a function call fails, the next binds short-circuit the processing rather than letting the next functions be evaluated in turn.

A third example is `seqn`, to transform a list of actions with side-effects (ex: IO) returning a result of type `a` into a single of such action:

```
seqn :: Monad m => [m a] -> m [a]
seqn []         = return []
seqn (act:acts) = do x <- act    -- *performs* this 'act' action
                     xn <- seqn acts
                     return (x:xs)
```

Properties

- any monad can be characterized as an adjunction between two (covariant) functors
- the monad as defined in category theory has been applied to functional programming, in order to provide semantics for the lambda calculus

Interest

- as monadic values represent explicitly not only computed values but also the effects that these evaluations trigger, a monadic expression can be freely replaced by its value (referential transparency), which enables the use of various optimisation approaches based on rewriting

- a monad captures, centralises and unifies once for all recurring schemes to integrate side effects that would be otherwise more difficult to handle (ex: with CPS, Continuation-Passing Style
- monads may register additional data that are inaccessible from functions and/or may drive their execution (ex: conditional call)
- monads may favour aspect-oriented programming, in the sense that they allow the developer to focus on his domain-specific logic (as the binding code, provided by the monad, is defined separately - and once for all)

Usages

- to condense code and to tie it to a mathematical formulation (compilation-time version of the "decorator pattern"
- to facilitate static analysis and program proofs
- to support the definition of simple DSL (*Domain-Specific Languages*) and to combine parsing rules
- to support the traversal of datastructures (see zipper)
- to transform complicated sequences of function calls into a compact pipeline abstracting out the management of additional data, flow control and side-effects
- to rely on call-by-need
- to allow for optimisations such as:
  - deforestation (a.k.a. fusion or "tree suppression"): a program transformation to eliminate intermediate lists or tree structures that are created and then immediately consumed by a program
  - memoisation: the caching of results of function calls in order to compute them once
  - parallelisation: to split a program between multiple logical processes
  - strong reduction: when -reduction in the -calculus are also performed on function bodies

See also this section about monad applications.

Lazy Evaluation

Most languages perform strict, eager evaluation: to evaluate a function call, first each of the supplied argument is fully evaluated, then the function itself is evaluated, based on the values jst computed for its arguments.

On the contrary, lazy evaluation strives to defer as much as possibly evaluations - possibly delaying up to the point of having never to perform them.

Lazy evaluation is convenient to express higher-level programs (ex: handling infinite lists such as `[1,...]`), yet makes it harder to predict the behaviour of programs at least in terms of resource consumption (time, memory, etc.).

For an increased control, strict evaluation can nevertheless be forced.

Arrows

An arrow (a.k.a. "bolt") is a type class representing computations in a pure way; this monad generalisation allows to express the relationships between the logical steps of a processing.

Refer to this article for more information.

Lambda Calculus

The lambda calculus is a simple yet powerful theory of functions. Its basis is to consider that all program elements are functions. An expression may include functions that are not yet defined and that are then considered as variables.

This is a formal system designed by Alonzo Church in the 1930s to define the concepts of function and (function) application. It relies on -expressions, where denotes the binding of a variable. For example, if `M` is a -expression, then `x.M` is also one, and represents the function that to the `x` variable associates `M` (i.e. `\x -> M x` here).

The -calculus has been the first formalism defining and characterising the recursive functions, and as such is essential to the theory of computing.

It is used as theoretical programming language and also as metalanguage for formal proof.

-calculus may or may not be typed.

Refer to this article for more information.

Minor Topics

- **enumerations**: ranges whose bounds (if any) are specified; any value in the `Enum` class can be used; ex: `[1..10]`, `[1,3..100]`, `['a' .. 'z']`, `[100..]`
- **operator sections**: if `#` is an operator, `(#)`, `(x #)`, `(# y)` are sections; for example `(/3)` ≡ `\x -> x/3`; usage: `sum = foldl (+) 0`
- **higher-order functions**: such a function takes a function as an argument and/or returns a function as a result; the support for currying in Haskell systematises the latter, so in this context a higher function is the latter, i.e a function with at least one argument being a function
- the `IO` monad accounts for the (input-output) side effects triggered by a program, which translate to changes done to the state of the world (outside of the program itself); such an effectful program, which is to return a value of type `a` can be represented by a `WorldState -> (a,WorldState)` function; this corresponds to the definition of the `IO a` type, whose values are called *actions* (i.e. transformations that may be executed)
- the `main` function of a script is evaluated when this script is executed; `main :: IO ()` returns the empty tuple, to be understood as a dummy result value; thanks to currying, no specific support for input program parameter is needed for the `IO` monad; for example an interactive program taking a character and returning an integer could have for type `Char -> IO Int`, meaning `Char -> WorldState -> (Int,WorldState)`

Haskell Syntax

Reserved Words

They cannot be used to name functions or variables:

```
case, class, data, deriving, do, else, if, import,
in, infix, infixl, infixr, instance, let, of, module,
newtype, then, type, where
```

Comments

A single-line comment starts with `--` and extends to the end of the line.

Multi-line comments start with `{-` and extend to `-}`.

Example:

```
size = 3  -- This is a constant here.

{- Comments are essential for understanding.

   Consider writing them.


 -}
```

Comments can be nested.

See also the comment conventions for the Haddock documentation generator.

Literate programming (where text is by default a comment, unless being specifically designated as code) is possible in an Haskell script, whose extension shall then be `.lhs` (for Literate Haskell).

Haskell Tools

Glasgow Haskell Tools

They include a compiler (`GHC`) and an interpreter (`GHCi`).

Haddock: a Haskell Documentation Tool

Haddock generates documentation from annotated Haskell source code (typically libraries).

So that they are taken into account by Haddock, comments above function definitions should start with `{- |`, and those next to parameter types with `-- ^`.

Example of use:

```
-- |The 'square' function squares an integer.
-- It takes one argument, of type 'Int'. square :: Int -> Int
square x = x * x

{-|
```

```
 The 'cube' function cubes an integer.
 It takes one argument, of type 'Int'.
 -}
cube x = x * square x

data T a b
  = C1 a b  -- ^ This is the documentation for the 'C1' constructor
  | C2 a b  -- ^ This is the documentation for the 'C2' constructor
```

See this page for more information regarding markup.

Haskell Conventions

Index

They start at zero:

```
[1,2,3,4] !! 1 = 2
```

Whitespaces

One should avoid tabulations (i.e. prefer spaces).

Generally the least number of spaces is preferred; like in: `sum $ map sqrt [1..10]`.

Naming

The names of functions and arguments start with a lowercase character and are followed by any number of: numbers, letters (of any case), underscores and single quotes.

The names of types and constructors start with an uppercase letter.

Shortness

Most Haskell developers seem to strongly dislike typing, often resulting in cryptic names for functions (ex: `fst`) and variable names (often reduced to a single character whose meaning is never disclosed).

A lot of efforts went especially to save keystrokes and more precisely to remove as much as possible the need for parentheses (function application being denoted as a space, the `.` and `$` operators being introduced, etc.). This allows very compact definitions of functions out of functions (ellipsing values as such), and an increased expressivity, sometimes at the expense of clarity. Extra documentation may alleviate this problem.

Single-letter variable names often denote their type (ex: `c` for a `Char` variable), and are suffixed by `s` to indicate this is a list thereof (ex: `bs` for a variable of type `[Bool]`, `css` for a `[[Char]]` one).

Layout & Indentation

Indentation matters, as spaces denote scopes:

```
a = b + c
 where
  b = 1
  c = 2

d = a * 2
```

An alternative layout based on curly braces and semi-colons exists, yet its use is discouraged.

A function body shall be indented of at least one space compared to the function name (if indented at all).

As any `where` or `let` use must be, indentation-wise, between the name and the body of a function, we prefer, compared to the function name:

- a 2-space indentation for the body
- a 1-space indentation for `where` / `let` body

For example:

```
square x =
  sq
 where sq = x * x
```

Haskell Hints

- generalise: types can be generalised thanks to type classes

Haskell In Practice

Installing Haskell

On Arch Linux (see this page for more information): `pacman -Sy ghc cabal-install stack`.

See also our corresponding script for continuous integration.

Running Haskell

Except for performances, programs can be tested directly with `ghci`, like in:

```
$ ghci Foobar.hs
```

The very essential GHCi commands are:

- `:?` or `:help`: lists available commands
- `:load FILENAME`: loads specified script

- `:reload` or `:r`: reloads the current module, file, or project
- `::` repeats the previous command
- `:type` or `:t`: returns the type of specified expression (value or function); ex: `:t not False`
- `set editor NAME`: sets the code editor to NAME
- `edit FILENAME`: edits specified script
- `edit`: edits current script
- `:quit` or `CTRL-D`: quits the interpreter

For example:

```
$ ghci
GHCi, version 8.10.5: https://www.haskell.org/ghc/  :? for help

Prelude> :type not
not :: Bool -> Bool
```

Functions can be directly redefined:

```
Prelude> fac n = product [1..n]
Prelude> fac 5
120

Prelude> fac _ = 0
Prelude> fac 5
0
```

Haskell-related Filenames

Filenames

They may or may not start with a capital letter (and no specified convention applies).

They may be in snake_case or in CamelCase.

Ex: `my_test.hs`, `HelloWorld.hs`.

Extensions

The main extensions are:

- `.hs`: Haskell source code (to preprocess then compile)
- `.lhs`: literate Haskell source (to unlit, preprocess and compile), where all text is comment by default, and code is specifically designated as such
- `.hi`: interface file; contains information about exported symbols
- `.hc`: intermediate C files

A very basic `Foobar.hs` source file once compiled results in a standard `Foobar.o` object file, typically:

```
ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

If `Foobar.hs` defines a suitable `main` function, once linked it results in a standard `Foobar.o` executable, typically (depending on the build options):

```
ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /
```

In terms of shared libraries, this boils down to, for example:

```
linux-vdso.so.1
libHSbase-4.14.2.0-ghc8.10.5.so => /usr/lib/ghc-8.10.5/base-4.14.2.0/libHSbase-4.14.2
libHSinteger-gmp-1.0.3.0-ghc8.10.5.so => /usr/lib/ghc-8.10.5/integer-gmp-1.0.3.0/libHS
libHSghc-prim-0.6.1-ghc8.10.5.so => /usr/lib/ghc-8.10.5/ghc-prim-0.6.1/libHSghc-prim-0
libHSrts-ghc8.10.5.so => /usr/lib/ghc-8.10.5/rts/libHSrts-ghc8.10.5.so
libgmp.so.10 => /usr/lib/libgmp.so.10
libc.so.6 => /usr/lib/libc.so.6
libm.so.6 => /usr/lib/libm.so.6
librt.so.1 => /usr/lib/librt.so.1
libdl.so.2 => /usr/lib/libdl.so.2
libffi.so.7 => /usr/lib/libffi.so.7
libpthread.so.0 => /usr/lib/libpthread.so.0
/lib64/ld-linux-x86-64.so.2 => /usr/lib64/ld-linux-x86-64.so.2
```

Build System

One may rely on the Haskell transposition of our simple, usual make-based build system (refer to the `GNUmake*` files; see this section of our Erlang-based Myriad counterpart for more details).

Haskell Resources

To Learn

We would certainly recommend browsing the pleasant Learn You a Haskell for Great Good! website or, even better, buying their book; another worthwhile book is *Programming in Haskell*, whose author is Graham Hutton, that we found interesting and well-written as well.

Of course the official Haskell website is also of interest.

Cheat Sheets

See Justin Bailey's one.

Support

Bugs, questions, remarks, patches, requests for enhancements, etc. are to be reported to the project interface (typically issues) or directly at the email address mentioned at the beginning of this cookbook.

Please React!

If you have information more detailed or more recent than those presented in this document, if you noticed errors, neglects or points insufficiently discussed, drop us a line! (for that, follow the Support guidelines).

Ending Word

Have fun with Haskell and functional programming!